Advanced list scheduling heuristic for task scheduling with communication contention for parallel embedded systems

Pengcheng Mu, Jean François Nezan, Mickael Raulet, Jean-Gabriel Cousin

SCIENCE CHINA

? 2010 Vol. ? No. ?: 1–14
doi:

Advanced List Scheduling Heuristic for Task Scheduling with Communication Contention for Parallel Embedded Systems

MU PengCheng[1]*, NEZAN Jean-François[2], RAULET Mickaël[2] & COUSIN Jean-Gabriel[2]

[1]*Ministry of Education Key Lab for Intelligent Networks and Network Security, School of Electronics and Information Engineering, Xi'an Jiaotong University, Xi'an* 710049*, China,*
[2]*IETR/Image and Remote Sensing Group, CNRS UMR 6164/INSA Rennes,* 35043 *RENNES Cedex, France*

Received September 18, 2009; accepted April 2, 2010

Abstract Modern embedded systems tend to use multiple cores or processors for processing parallel applications. This paper indeed aims at task scheduling with communication contention for parallel embedded systems and proposes three advanced techniques to improve the list scheduling heuristic. Five groups of node levels *(two existing groups and three new groups)* are firstly used as node priorities to generate node lists. Then the *critical child* technique improves the selection of a processor in the scheduling process. Finally, the *communication delay* technique enlarges the idle time intervals on communication links. We also propose an advanced dynamic list scheduling heuristic by combining the three techniques. Experimental results show that the combined advanced dynamic heuristic is efficient to shorten the schedule length for most of the randomly generated DAGs in the cases of medium and high communication. Our method accelerates an application up to 80% in the case of high communication and can also reduce the use of hardware resources.

Keywords list scheduling, communication contention, node level, critical child, communication delay

Citation Mu P C. Advanced List Scheduling Heuristic for Task Scheduling with Communication Contention for Parallel Embedded Systems.

1 Introduction

The recent evolution of digital communication and video compression applications has dramatically increased complexities of both the algorithm and the embedded system. To face this problem, System on a Chip (SoC), which embeds several cores (e.g. multi-core DSPs) and several hardware accelerators (e.g. Intellectual Properties), becomes the basic element to build complex embedded systems; and dataflow programming has been proposed for multiprocessor programming[1]. Task scheduling of a dataflow program over a multi-component embedded system is becoming more and more important due to the growing requirements of applications. However, task scheduling is not straightforward; when performed manually, the result is usually a suboptimal solution. Scheduling on general parallel computer architectures has been actively researched, but task scheduling on parallel embedded systems[2] is different from the general scheduling problem. Communications between cores have a very important impact on the scheduling and the resulting use of the hardware resources. Hence, it is necessary to find new task scheduling methodologies which produce optimal or near optimal results for parallel embedded systems.

In the task scheduling problem, the program is represented as a task graph modeled by Directed Acyclic Graph (DAG)[2,3], where nodes represent tasks (i.e. computations) and edges represent dataflows (i.e. communications) between tasks. The objective of task scheduling is to respectively assign computations and communications to

*Corresponding author (email: pengchengmu@gmail.com)

processors and buses (communication links) of the target system in order to get the minimum schedule length (makespan). The scheduling could be static (done at compile time) or dynamic (done at run time). Static scheduling is more suitable than dynamic scheduling for deterministic applications in parallel embedded systems by leading to lower code size and higher computation efficiency. This paper tackles the static scheduling problem for programming on parallel embedded systems, and all the task scheduling heuristics in the following parts are done at compile time.

The general task scheduling problem is proven to be NP-hard[3,4]; hence, many works try to find heuristics to go up to the optimal solution. Early task scheduling heuristics do not consider communications between tasks[5,6]. As communications increase in modern applications, many scheduling heuristics have to take them into account[3,7−10]. Most of these heuristics use fully connected topology structures of systems in which all communications can be concurrently performed. Different arbitrary processor networks are then used in refs. [11-15] to accurately describe real parallel systems, and the task scheduling takes into account communication contentions on communication links.

Most of the above heuristics are based on the approach of list scheduling. Basic techniques are given in ref. [16] for list scheduling with communication contention. This paper will give an advanced list scheduling heuristic with several advanced techniques for task scheduling with communication contention in parallel embedded systems. Three new groups of node levels will be firstly defined and used as node priorities to generate node lists in addition to the two existing groups; secondly, a technique of using a node's ***critical child*** will be given to improve the performance for selecting a processor for a node; and thirdly, the ***communication delay*** technique delays a communication when necessary in order to enlarge idle time intervals on communication links. This paper will finally combine these three techniques and show the efficiency of the results.

The paper is organized as follows: Section 2 firstly introduces the necessary models and definitions, then the task scheduling problem with communication contention is described in this section. Different node levels are given in section 3 by considering the communication contention. Section 4 gives the list scheduling heuristics including the classic static heuristic and our advanced heuristic. Experimental results to compare our heuristic to the classic one are given in section 5. The paper is concluded in section 6.

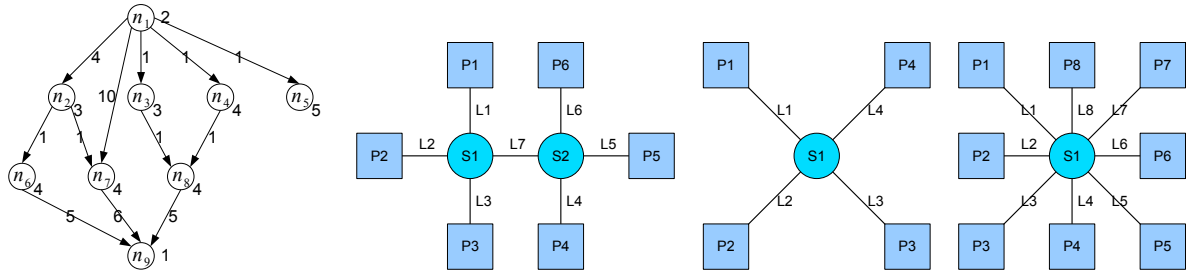
2 Models and Definitions

The program to be scheduled is called an algorithm and is modeled as a DAG in this paper. The multiprocessor parallel embedded system is called an architecture and is modeled as a topology graph. These two models are detailed as follows.

2.1 DAG Model

A DAG is a directed acyclic graph $G = (V, E, w, c)$ where V is the set of nodes and E is the set of edges. For two nodes $n_i, n_j \in V$, e_{ij} denotes the edge from the origin node n_i to the destination node n_j. A node represents a computation, and the weight of node n_i (denoted by $w(n_i)$) represents the time cost of computation. An edge represents the communication between two nodes, and the weight of edge e_{ij}(denoted by $c(e_{ij})$) represents the time cost of communication. In this model, the set $\{n_x \in V : e_{xi} \in E\}$ of all the direct predecessors of node n_i is denoted by $pred(n_i)$; the set $\{n_x \in V : e_{ix} \in E\}$ of all the direct successors of node n_i is denoted by $succ(n_i)$. A node n_i with $pred(n_i) = \emptyset$ is named a source node, and a node n_i with $succ(n_i) = \emptyset$ is named a sink node, where $\emptyset$ is the empty set.

The execution of computations on a processor is sequential. A computation can not be divided into parts. A computation can not start until all its input communications finish; all its output communications can not start until this computation finishes. Communications are also sequential on a communication link, but different computations and communications can be executed simultaneously respecting the input and output constraints given above. Figure 1(a) gives a DAG example used in ref. [17] to illustrate performances of different scheduling heuristics. It is also used in subsection 5.1 to show the performance of our method.



(a) Algorithm (b) Architecture 1 (c) Architecture 2 (d) Architecture 3

Figure 1: System models

2.2 Topology Graph Model

A topology graph $TG = (N, P, D, H, b)$ has been used to model a target system of multiple processors interconnected by communication links and switches[14]. N is the set of vertices; P is a subset of N, $P \subseteq N$; D is the set of directed edges; H is the set of hyperedges; b is the relative data rate of edge. The union of the two edge sets D and H is designated the link set L, $L = D \cup H$; an element of this set is denoted by $l \in L$. The topology graph is also denoted by $TG = (N, P, L, b)$.

Since a parallel embedded system usually consists of multiple heterogenous components, the topology graph is used to model it in this paper. A vertex $p \in P$ represents a processor; a vertex $n \in N, n \notin P$ represents a switch. It is supposed that directed edges are not used in a topology graph. Hence, a link $l \in L$ is actually a hyperedge h, which is a subset of two or more vertices of N, $h \subseteq N, |h| > 1$. A hyperedge connects multiple vertices and represents a half duplex multidirectional communication link (e.g. a bus). The weight $b(l)$ associated with a link $l \in L$ represents its relative data rate.

Differing from the processor, a switch is an ideal vertex only used for connecting communication links, and no computation can be executed on it.

Ideal Switch: For a switch s, let $l_1, l_2, \ldots, l_n$ be all the communication links connected to s. If two links l_{i_1} and l_{i_2} of them are not used for the moment, a communication can be transferred from l_{i_1} to l_{i_2} without any impact from/to communications on other communication links connected to s.

Figure 1(b) gives an architecture example with six processors ($P1$, $P2$, $P3$, $P4$, $P5$ and $P6$) interconnected by seven links ($L1$, $L2$, $L3$, $L4$, $L5$, $L6$ and $L7$) and two switches ($S1$ and $S2$). This architecture models TI's C6474 Evaluation Module (EVM) which includes two C6474 multicore DSPs[1]. Figure 1(c) and 1(d) also show two other architectures which will be used for the experimental results in subsection 5.1 and 5.2, respectively.

A route is used to transfer data from one processor to another in a parallel embedded system. It is a chain of links connected by switches from the origin processor to the destination processor. For example, $L1 \rightarrow L7 \rightarrow L4$ is a route from $P1$ to $P4$ in Figure 1(b). A link l on a route R is denoted by $l \in R$. All the routes from processor p_i to processor p_j compose a set of routes $RS(p_i, p_j)$. If $p_i = p_j$, then $RS(p_i, p_j) = \emptyset$, which means no route is needed.

Routing is a procedure of generating routes and is an important aspect of task scheduling. In ref. [15], the route is dynamically created during the scheduling to improve the performance, but it does not use switches in the system architecture. In fact, routes are usually determined once and stored in a table for parallel embedded systems using switches, which means static routing. This paper uses the static routing and supposes that there is at least a route between any two processors. Hence, the routing during the scheduling becomes looking up the table of routes.

2.3 Task Scheduling with Communication Contention

A schedule of a DAG is the association of a start time and a processor with each node of the DAG. When the communication contention is considered, a schedule also includes allocating communications to links and associating start times on these links with each communication. A schedule S of a DAG $G = (V, E, w, c)$ over a topology graph $TG = (N, P, L, b)$ is described by the following terms.

[1] http://www.ti.com/

The start time of a node $n_i \in V$ on a processor $p \in P$ is denoted by $t_s(n_i, p)$; the finish time is given by

$$t_f(n_i, p) = t_s(n_i, p) + w(n_i, p)$$

where $w(n_i, p)$ is the execution duration of n_i on p. The schedule length of S is the maximum finish time among all the nodes,

$$sl(S) = \max_{n_i \in V} \{t_f(n_i, proc(n_i))\}$$

where $proc(n_i)$ denotes the processor on which n_i is allocated.

Since execution durations of a node on different processors can be very different ($w(n_i, p_j) \gg w(n_i, p_k)$), this node is usually constrained to some processors which give relatively small execution durations. The set of processors on which n_i can be executed is denoted by $Proc(n_i)$. The average computation duration of a node on different processors is used to represent the node weight which is given by

$$w(n_i) = \frac{1}{|Proc(n_i)|} \sum_{p \in Proc(n_i)} w(n_i, p)$$

where $|Proc(n_i)|$ is the number of processors in $Proc(n_i)$.

The communication represented by an edge is needed only when the edge's origin node and destination node are not allocated on the same processor. The start time of an edge $e_{ij} \in E$ on a link l of route R is denoted by $t_s(e_{ij}, l, R)$. Communications are handled in the way of cut-through on a route because of the use of circuit switching in embedded systems. Hence, e_{ij} is aligned on all the links of the route $R = l_1 \to l_2 \to \ldots \to l_k$ with $t_s(e_{ij}, l_1, R) = t_s(e_{ij}, l_2, R) = \ldots = t_s(e_{ij}, l_k, R)$. The route on which e_{ij} is allocated is denoted by $R(e_{ij})$. The start time and finish time of e_{ij} on all the links of the route $R = R(e_{ij})$ are uniformly denoted by $t_s(e_{ij}, R)$ and $t_f(e_{ij}, R)$ with $t_f(e_{ij}, R) = t_s(e_{ij}, R) + \frac{d(e_{ij})}{\min\limits_{l \in R}\{b(l)\}}$, where $d(e_{ij})$ is the number of data to be transferred by e_{ij}, and $\min\limits_{l \in R}\{b(l)\}$ is the minimum data rate of the links in the route R. The average communication duration of an edge on all its possible routes is used to represent the edge weight which is given by

$$c(e_{ij}) = \frac{1}{\sum\limits_{p_x, p_y} |RS(p_x, p_y)|} \sum_{p_x, p_y} \left\{ \sum_{R \in RS(p_x, p_y)} \frac{d(e_{ij})}{\min\limits_{l \in R}\{b(l)\}} \right\}$$

where $p_x \in Proc(n_i), p_y \in Proc(n_j)$. This kind of calculation for $c(e_{ij})$ is firstly proposed in this paper and is more suitable for task scheduling in parallel embedded systems.

A node (computation) can start on a processor at the time when all the node's input edges (communications) finish. This time is called the Data Ready Time (DRT) and is denoted by

$$DRT(n_j, p) = \max_{e_{ij} \in E} \{t_f(e_{ij}, R(e_{ij}))\}$$

DRT is the earliest time when a node can start. If n_j is a node without input edge, then $DRT(n_j, p) = 0, \forall p \in P$, which means data of n_j are ready at the beginning (time 0).

The insertion technique is usually used for node and edge scheduling[16]. The conditions to use the insertion technique for node and edge scheduling are explained as follows.

Node Scheduling Condition: For a node n_i, let $[A, B]$ ($A, B \in [0, \infty]$) be an idle time interval on the processor p. n_i can be scheduled on p within $[A, B]$ if $\max\{A, DRT(n_i, p)\} + w(n_i, p) \leqslant B$. The start time of n_i on p is given by $t_s(n_i, p) = \max\{A, DRT(n_i, p)\}$.

Edge Scheduling Condition: For an edge e_{ij}, let R be a route for this edge and let $[A, B]$ ($A, B \in [0, \infty]$) be a common idle time interval on all the links of this route. e_{ij} can be scheduled on R within $[A, B]$ if $\max\{A, t_f(n_i, proc(n_i))\} + \frac{d(e_{ij})}{\min\limits_{l \in R}\{b(l)\}} \leqslant B$. The start time of e_{ij} on this route is given by $t_s(e_{ij}, R) = \max\{A, t_f(n_i, proc(n_i))\}$.

3 Node Levels with Communication Contention

The top level and bottom level are usually used as node priorities which are important for DAG scheduling[11,18]. The top level of a node is the length of the longest path from any source node to this node, excluding the weight of this node; the bottom level of a node is the length of the longest path from this node to any sink node, including the weight of this node. Two groups of top and bottom levels have been used in task scheduling heuristics, which are: 1) computation top and bottom levels (tl_{comp} and bl_{comp}), 2) top and bottom levels (tl and bl). In addition, this paper proposes three new groups which are named as: 3) input top and bottom levels (tl_{in} and bl_{in}), 4) output top and bottom levels (tl_{out} and bl_{out}), 5) input/output top and bottom levels (tl_{io} and bl_{io}). Figure 2 illustrates the dependencies between nodes to define different top levels and bottom levels, where the red dotted nodes and edges are used to recursively define the top levels and bottom levels of n_i.

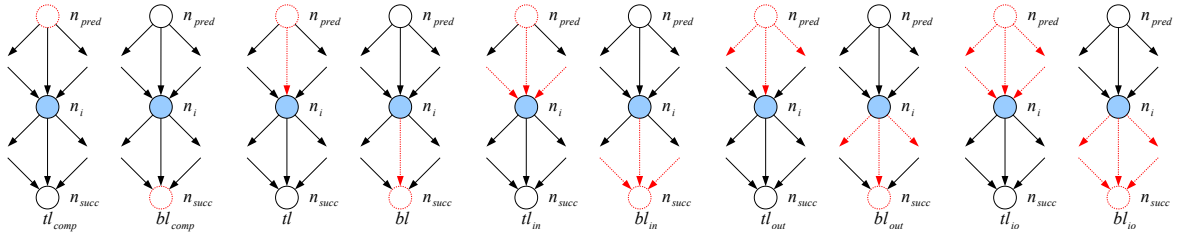


Figure 2: Five groups of node levels

1. Computation top level and bottom level (Figure 2(a))

   The computation top level of a node is the length of the longest path from any source node to this node only including the weights of nodes; the computation bottom level of a node is the length of the longest path from this node to any sink node only including the weights of nodes. The weights of edges are not taken into account in the computation top level and bottom level. They are recursively defined as follows:

$$tl_{comp}(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \{tl_{comp}(n_k) + w(n_k)\}, & \text{otherwise} \end{cases}$$

$$bl_{comp}(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \{bl_{comp}(n_k)\} + w(n_i), & \text{otherwise} \end{cases}$$

2. Top level and bottom level (Figure 2(b))

   The top level and bottom level additionally take into account the weights of edges on the path by contrast with the computation top level and bottom level. They are recursively defined as follows:

$$tl(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \{tl(n_k) + w(n_k) + c(e_{ki})\}, & \text{otherwise} \end{cases}$$

$$bl(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \{bl(n_k) + c(e_{ik})\} + w(n_i), & \text{otherwise} \end{cases}$$

3. Input top level and bottom level (Figure 2(c))

   The input top level and bottom level take into account weights of nodes on the path as well as weights of all the input edges of a node on the path. They are recursively defined as follows:

$$tl_{in}(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \{tl_{in}(n_k) + w(n_k)\} + \sum\limits_{e_{li} \in E} c(e_{li}), & \text{otherwise} \end{cases}$$

$$bl_{in}(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \left\{ bl_{in}(n_k) + \sum\limits_{e_{lk} \in E} c(e_{lk}) \right\} + w(n_i), & \text{otherwise} \end{cases}$$

4. Output top level and bottom level (Figure 2(d))

   The output top level and bottom level take into account weights of nodes on the path as well as weights of all the output edges of a node on the path. They are recursively defined as follows:

$$tl_{out}(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \left\{ tl_{out}(n_k) + w(n_k) + \sum\limits_{e_{kl} \in E} c(e_{kl}) \right\}, & \text{otherwise} \end{cases}$$

$$bl_{out}(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \{bl_{out}(n_k)\} + \sum\limits_{e_{il} \in E} c(e_{il}) + w(n_i), & \text{otherwise} \end{cases}$$

5. Input/output top level and bottom level (Figure 2(e))

   The input/output top level and bottom level take into account weights of nodes on the path as well as weights of all the input and output edges of a node on the path. They are recursively defined as follows:

$$tl_{io}(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \left\{ tl_{io}(n_k) + w(n_k) + \sum\limits_{e_{kl} \in E} c(e_{kl}) - c(e_{ki}) \right\} + \sum\limits_{e_{li} \in E} c(e_{li}), & \text{otherwise} \end{cases}$$

$$bl_{io}(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \left\{ bl_{io}(n_k) + \sum\limits_{e_{lk} \in E} c(e_{lk}) - c(e_{ik}) \right\} + \sum\limits_{e_{il} \in E} c(e_{il}) + w(n_i), & \text{otherwise} \end{cases}$$

The three new groups take into account the communication contention between nodes in comparison with the two existing groups which are usually used in the list scheduling without communication contention. Table 1 gives all the five groups of top levels and bottom levels for the DAG given in Figure 1(a). This table will be used in subsection 5.1.

Table 1: Different node levels

	tl_{comp}	bl_{comp}	tl	bl	tl_{in}	bl_{in}	tl_{out}	bl_{out}	tl_{io}	bl_{io}
n_1	0	11	0	23	0	41	0	35	0	55
n_2	2	8	6	15	6	35	19	16	19	36
n_3	2	8	3	14	3	26	19	14	19	26
n_4	2	9	3	15	3	27	19	15	19	27
n_5	2	5	3	5	3	5	19	5	19	5
n_6	5	5	10	10	10	21	24	10	24	21
n_7	5	5	12	11	20	21	24	11	34	21
n_8	6	5	8	10	9	21	24	10	25	21
n_9	10	1	22	1	40	1	34	1	54	1

4 List Scheduling Heuristics

List scheduling is an important task scheduling heuristic. Algorithm 1 gives a commonly used static list scheduling heuristic as given in ref. [14].

This algorithm consists of three procedures. Nodes are firstly sorted into a static list by the procedure of `Sort_Nodes()` in the heuristic, then a processor is selected for each node by `Select_Processor()` and this node is scheduled by `Schedule_Node()`. Since the order of nodes in the list affects the schedule result, many different priority schemes have been proposed to sort nodes[10,11]. Experiments in ref. [18] show that list scheduling with static list sorted by bottom level outperforms other compared contention aware algorithms. Hence, this paper uses the following rule to sort nodes.

Rule of Sorting Nodes: Nodes are sorted by the decreasing order of their bottom levels; if two nodes have equal bottom levels, the one with greater top level is placed before the other; if both the bottom level and the top level are equal, these nodes are randomly sorted.

Algorithm 1: `Static_List_Scheduling`(G, TG)

Input: A DAG $G = (V, E, w, c)$ and a topology graph $TG = (N, P, L, b)$
Output: A schedule of G on TG

```
1 NodeList ← Sort_Nodes(V);
2 for each n ∈ NodeList do
3 |   p_best ← Select_Processor(n, P);
4 |   Schedule_Node(n, p_best);
5 end
```

Details about the static list scheduling heuristic can be found in ref. [16]. This heuristic is considered as a classic list scheduling heuristic and will be used for the comparison with our advanced method. The following gives two advanced list scheduling techniques and an advanced dynamic list scheduling heuristic using these two techniques.

4.1 Processor Selection with Critical Child

The classic list scheduling heuristic selects the processor allowing the earliest finish time for a node. This rule probably gives a locally optimized result. In fact, this rule usually gives bad results for the join structure of a DAG especially in the case of great communication cost and communication contention. Figure 3(a) shows such an example; Figure 3(b) gives the schedule result with the classic processor selection method, which selects a new processor for each one of n_1, n_2 and n_3 to provide the earliest finish time. Hence, the execution of node n_4 has to wait until the communications from n_2 and n_3 finish, and the schedule length is 6 at last. By contrast, the schedule of all nodes on the same processor is shown in Figure 3(c) and has a schedule length of 4. The reason for the bad result of the classic method is that the successor is not taken into account during the processor selection, hence we propose a technique of critical child to avoid this bad result.

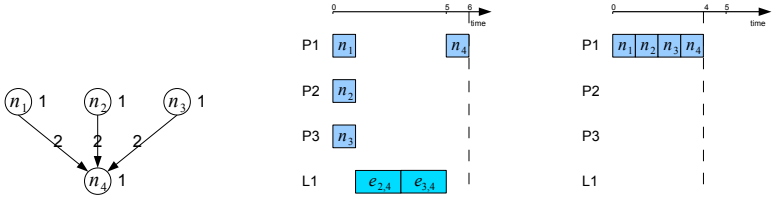


Figure 3: A join DAG and two different schedule results

In ref. [10], the critical child of a node is defined as one of its successors that has the smallest difference between the absolute latest possible start time (ALST) and the absolute earliest possible start time (AEST). It is used for scheduling in the case of unbounded number of processors and without communication contention. We use the concept of critical child for list scheduling in the case of bounded number of processors and with communication contention. The critical child is differently defined as follows.

Critical Child: Given a static node list $NodeList$, the critical child of node n_i is denoted by $cc(n_i)$ and is one of n_i's successors that firstly emerges in $NodeList$.

According to this definition, the critical child of n_i may be different if $NodeList$ differs though the DAG is not changed. This is the difference between our critical child and that in ref. [10]. Using critical child makes the processor selection take into account not only the predecessors of a node, but also its most important successor. Our method of using the critical child to select processor is given in Algorithm 2.

An unscheduled node with all its predecessors having been scheduled is called a free node. Since it is possible that $cc(n_i)$ is not a free node during the processor selection for n_i, the scheduling of $cc(n_i)$ only takes into account the critical child's scheduled predecessors in the procedure of `Select_Processor()`, which will be shown in the algorithm of edge scheduling.

4.2 Node and Edge Scheduling with communication delay

Our methods of node and edge scheduling differ from those of the classic one by using the As Late As Possible (ALAP) start time to delay communications. Given the route $R = l_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_k$ for edge e_{ij}, let e_m be the

Algorithm 2: `Select_Processor`(n_i, P)

Input: A node $n_i \in V$ and the set P of all the processors
Output: The best processor p_{best} for the input node n_i

```
1  Find the critical child cc(n_i);
2  BestFinishTime ← ∞;
3  for each p ∈ Proc(n_i) do
4      FinishTime ← Schedule_Node(n_i, p, true);
5      MinFinishTime ← ∞;
6      if cc(n_i) ≠ null then
7          for each p' ∈ Proc(cc(n_i)) do
8              FinishTime ← Schedule_Node(cc(n_i), p', true);
9              MinFinishTime ← min{MinFinishTime, FinishTime};
10             Unschedule the input edges of cc(n_i);
11             Unschedule cc(n_i) from p';
12         end
13     else
14         MinFinishTime ← FinishTime;
15     end
16     if MinFinishTime < BestFinishTime then
17         BestFinishTime ← MinFinishTime;
18         p_best ← p;
19     end
20     Unschedule the input edges of n_i;
21     Unschedule n_i from p;
22 end
```

edge before which e_{ij} is scheduled on link l_m, the ALAP of e_{ij} is defined as

$$ALAP\left(e_{ij}\right) = \min\left\{t_s\left(e_1, R\left(e_1\right)\right), t_s\left(e_2, R\left(e_2\right)\right), \ldots, t_s\left(e_k, R\left(e_k\right)\right), t_s\left(n_j, proc\left(n_j\right)\right)\right\} - \frac{d(e_{ij})}{\min\limits_{l \in R}\{b(l)\}}$$

If e_m does not exist, which means e_{ij} is the last edge scheduled on l_m, then $t_s\left(e_m, R\left(e_m\right)\right) = \infty$.

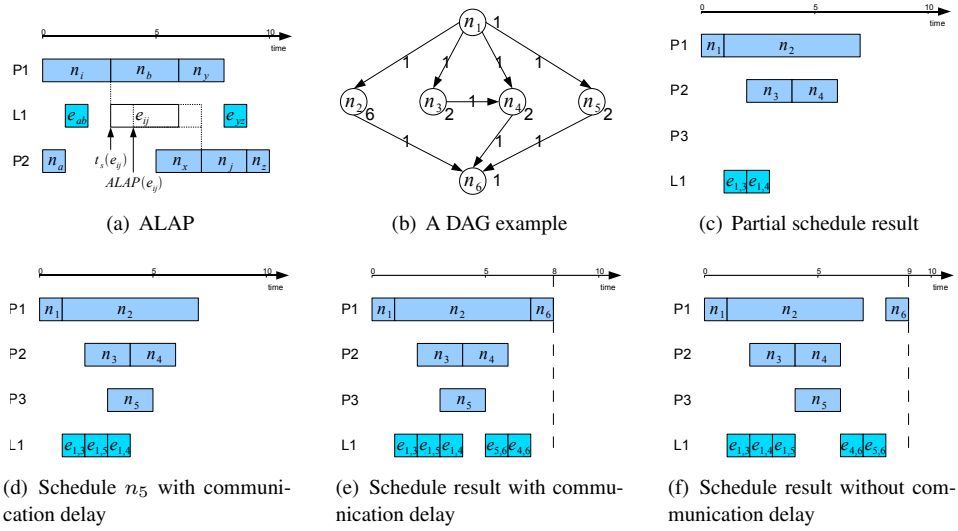
The communication can be delayed by using the ALAP, hence, an idle time interval is enlarged on the link. The idle time interval changes from $\left[t_f\left(e_{n-1}, R\left(e_{n-1}\right)\right), t_s\left(e_n, R\left(e_n\right)\right)\right]$ to $\left[t_f\left(e_{n-1}, R\left(e_{n-1}\right)\right), ALAP\left(e_n\right)\right]$ between two successive edges e_{n-1} and e_n on link l. If e_n is the first edge on link l, then $t_f\left(e_{n-1}, R\left(e_{n-1}\right)\right) = 0$; and if e_{n-1} is the last edge on link l, then $t_s\left(e_n, R\left(e_n\right)\right) = ALAP\left(e_n\right) = \infty$.

Figure 4(a) shows the use of ALAP. If e_{ij} is delayed to its ALAP, the idel time interval on $L1$ between e_{ab} and e_{ij} will be enlarged and a greater communication can be inserted bewteen e_{ab} and e_{ij}.

The method of scheduling a node n_i onto a processor p is given in Algorithm 3. When a node is scheduled, the ALAPs of its input edges are then calculated (line 6 to 10 in Algorithm 3). The ALAP of an edge can not be calculated during the processor selection. Hence, a Boolean value is used to indicate whether the procedure `Schedule_Node()` is used in the procedure `Select_Processor()` or not.

Algorithm 4 gives our method for edge scheduling which is similar to that of the classic heuristic. However, there is also some improvements: The origin node n_i of e_{ij} is tested because some predecessors of the critical child may be non-scheduled; the best route is chosen to give the earliest finish time; and the ALAP is considered in the edge scheduling condition.

Figure 4(b) gives a DAG example to show the effect of communication delay. Nodes are sorted into a static list of n_1, n_2, n_3, n_4, n_5, n_6 by using the priority of bl & tl. Figure 4(c) gives a partial schedule result with n_1, n_2, n_3, n_4 having been scheduled. As to n_5, the input edge $e_{1,4}$ for n_4 can start at its ALAP of time 3. Hence, the edge $e_{1,5}$ is inserted between $e_{1,3}$ and $e_{1,4}$ as shown in Figure 4(d) and finally a schedule length of 8 is obtained in Figure 4(e). If ALAP is not used, another schedule result is obtained in Figure 4(f) with the schedule length of 9.



(a) ALAP (b) A DAG example (c) Partial schedule result

(d) Schedule n_5 with communication delay (e) Schedule result with communication delay (f) Schedule result without communication delay

Figure 4: Communication delay

Algorithm 3: `Schedule_Node`(n_i, p, $IsTemporary$)

Input: $n_i \in V$, a processor $p \in Proc(n_i)$ and a Boolean value $IsTemporary$

Output: The finish time of n_i on p

```
for each n_l ∈ pred(n_i), proc(n_l) ≠ p do
    Schedule_Edge(e_li, p);
end
Calculate DRT of node n_i;
Find the earliest idle time interval for node n_i on processor p respecting the node scheduling condition;
if IsTemporary = false then
    for each n_l ∈ pred(n_i), proc(n_l) ≠ p do
        Calculate the ALAP of e_li;
    end
end
Schedule n_i on p and calculate the finish time;
```

4.3 Advanced Dynamic List Scheduling

Algorithm 5 shows our advanced dynamic list scheduling heuristic. The "dynamic" means that the node list is not determined before the scheduling but created during the scheduling. Hence, the procedure `Sort_Nodes()` for the static list scheduling heuristic is no longer necessary. The procedure `Choose_Node()` is used in place of the procedure `Sort_Nodes()` to choose a node for scheduling. The procedures `Select_Processor()` and `Schedule_Node()` use the new method given above.

As used for sorting nodes into static lists, node levels are also effective to create dynamic node lists. In the dynamic list scheduling, any free node can be scheduled in the next step, but we should choose the most critical one. Since the length of the longest path passing a free node during the scheduling is crucial to the final schedule length, the free node in this path, which is the first unscheduled node in this path, must be treated immediately in order to be executed as early as possible. This node is named the critical node and is obtained by considering the bottom level as shown in Algorithm 6.

In this algorithm, the bottom level ($bl(n_i)$) is used as the node priority. The bottom level reflects the time needed from this node to the end of the DAG; our new bottom levels reflect better the reality in the case of communication contention. Hence, $bl(n_i)$ can be replaced by other bottom levels like $bl_{comp}(n_i)$, $bl_{in}(n_i)$, $bl_{out}(n_i)$ and $bl_{io}(n_i)$. Different bottom levels may give different dynamic node lists and can finally lead to different schedule results.

Algorithm 4: `Schedule_Edge`(e_{ij}, p)

Input: $e_{ij} \in E$ and a processor $p \in Proc(n_j)$ on which the node n_j to be scheduled
Output: None

if n_i *is scheduled* **then**
  if $proc(n_i) \neq p$ **then**
    $FinishTime \leftarrow \infty$;
    for *each* $R \in RS(proc(n_i), p)$ **do**
      Find the earliest common idle time interval on all the links of R respecting the edge scheduling condition with ALAP;
      if $t_f(e_{ij}, R) < FinishTime$ **then**
        $FinishTime \leftarrow t_f(e_{ij}, R)$;
        $R_{best} \leftarrow R$;
      end
    end
    Schedule e_{ij} on R_{best};
  end
end

Algorithm 5: `Dynamic_List_Scheduling`(G, TG)

Input: A DAG $G = (V, E, w, c)$ and a topology graph $TG = (N, P, L, b)$
Output: A schedule of G on TG

$UNS \leftarrow V$;
while *existing nodes in* UNS **do**
  $n \leftarrow$ `Choose_Node`(UNS);
  $p_{best} \leftarrow$ `Select_Processor`(n, P);
  `Schedule_Node`(n, p_{best}, $false$);
  Remove n from UNS;
end

5 Experimental Results

This section gives experimental results of our proposed list scheduling heuristics compared to the classic one given in ref. [14]. The architecture in Figure 1(c) and 1(d) are used for the comparison in subsection 5.1 and 5.2, respectively.

5.1 Comparison with an Example

The DAG given in Figure 1(a) is used in this section to show the improvement by using the advanced dynamic heuristic with different node priorities. Table 1 has given all the five groups of top levels and bottom levels for this DAG, the resulting static lists according to the rule of sorting nodes are given in Table 2 which also shows the critical children according to these different static node lists.

Figure 5 gives the schedule result of the classic static list scheduling heuristic with nodes sorted by $bl\&tl$. In this figure, two different symbols for an edge respectively represent the sending and receiving of this edge. The classic heuristic gives the schedule length of 21.

Our advanced dynamic heuristic with different node priorities may give different dynamic node lists and finally gives different schedule results. Table 3 shows the generated dynamic node lists with the five node priorities, and it is noticed that four different node lists (from (a) to (d)) are obtained.

The schedule result for the node priority bl_{comp} is shown in Figure 6(a). The schedule length of 18 is obtained by using 3 processors. The schedule result for the node priority bl is shown in Figure 6(b), and the schedule length is also 18 with 3 processors. Figure 6(c) shows the schedule result with the node priority bl_{in}. The schedule length is also 18 but with 4 processors. Figure 6(d) gives the schedule result for the same node list obtained by bl_{out} and

Algorithm 6: `Choose_Node`(UN)

Input: A set UN of all the unscheduled nodes
Output: The critical node n_c among all the unscheduled nodes

Create a set FN of all the free nodes from UN;
$MaxLength \leftarrow 0$;
for *each* $n_i \in FN$ **do**
  $Length \leftarrow 0$;
  for *each* $n_l \in pred(n_i)$ **do**
    $Length \leftarrow \max\{Length, t_f(n_l, proc(n_l)) + bl(n_i)\}$;
  end
  if $MaxLength < Length$ **then**
    $MaxLength \leftarrow Length$;
    $n_c \leftarrow n_i$;
  else if $MaxLength = Length$ **then**
    if $bl(n_c) < bl(n_i)$ **then**
      $n_c \leftarrow n_i$;
    end
  end
end

Table 2: Different static node lists and corresponding critical children

Node priority	Static node list	Critical child								
		n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9
bl_{comp} & tl_{comp}	$n_1, n_4, n_3, n_2, n_8, n_7, n_6, n_5, n_9$	n_4	n_7	n_8	n_8	null	n_9	n_9	n_9	null
bl & tl	$n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null
bl_{in} & tl_{in}	$n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null
bl_{out} & tl_{out}	$n_1, n_2, n_4, n_3, n_7, n_8, n_6, n_5, n_9$	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null
bl_{io} & tl_{io}	$n_1, n_2, n_4, n_3, n_7, n_8, n_6, n_5, n_9$	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null

bl_{io}. The schedule length is 17 with 4 processors and is better than the three former schedule lengths of 18. All the schedule results of the advanced dynamic heuristic are better than that of the classic heuristic; sometimes the number of used processors is also reduced.

5.2 Comparison with Random DAG

Random graphs are commonly used to compare scheduling algorithms in order to get statistical results which are more persuasive than the result for some particular graphs. We implement a graph generator based on SDF^3 to generate random SDF graphs[19] except that the SDF graphs are constrained to be DAGs (no cycles).

A random DAG is constrained in five aspects: (1) the number of nodes, (2) the average in degree, (3) the average out degree, (4) the random weights of nodes, (5) the random weights of edges. The average in degree and out degree are assumed to be same in this paper. The weights of nodes vary randomly from w_{min} to w_{max}. The communication to computation ratio (CCR) is used to generate random weights of edges. The CCR is defined

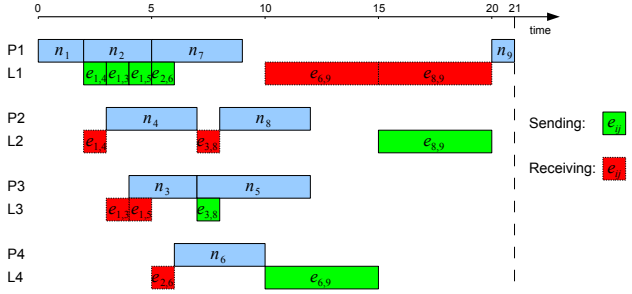


Figure 5: Schedule result of classic heuristic

Table 3: Different dynamic node lists

Node priority	Dynamic node list	No.
bl_{comp}	$n_1, n_4, n_2, n_6, n_7, n_3, n_8, n_9, n_5$	(a)
bl	$n_1, n_4, n_2, n_7, n_6, n_3, n_8, n_9, n_5$	(b)
bl_{in}	$n_1, n_2, n_4, n_3, n_8, n_6, n_7, n_9, n_5$	(c)
bl_{out}	$n_1, n_2, n_4, n_3, n_8, n_7, n_6, n_9, n_5$	(d)
bl_{io}	$n_1, n_2, n_4, n_3, n_8, n_7, n_6, n_9, n_5$	(d)

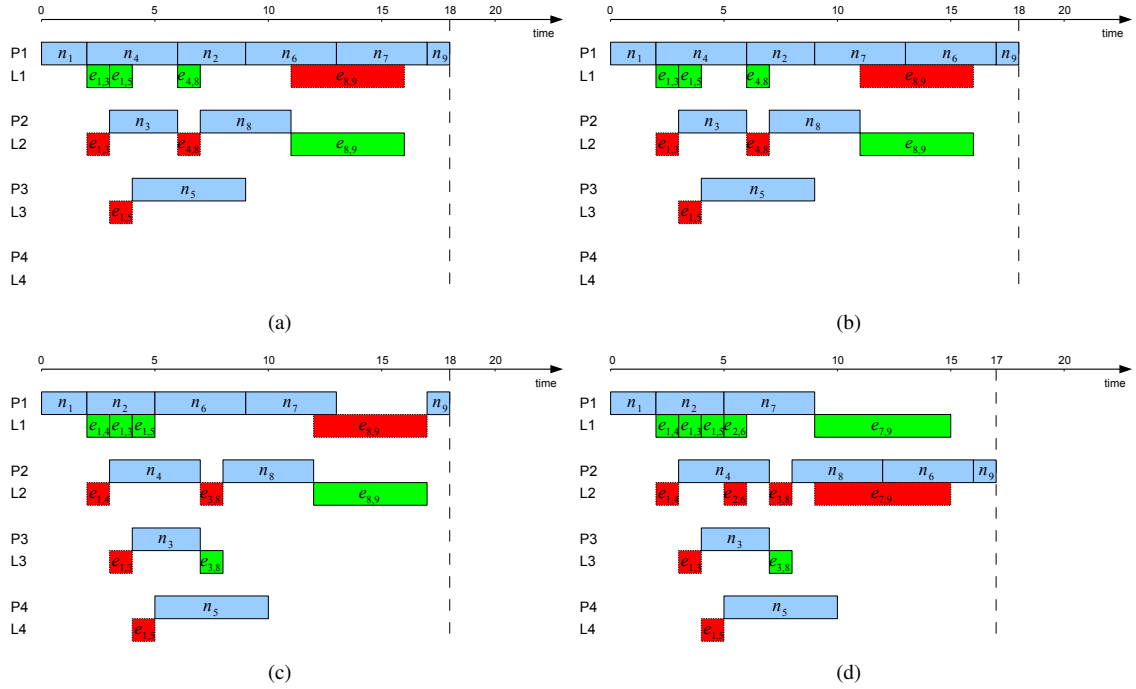


Figure 6: Schedule results of advanced dynamic heuristic

as the average weight of edges divided by the average weight of nodes in this paper, that is, $CCR = \frac{\frac{1}{|E|}\sum_{e\in E} c(e)}{\frac{1}{|V|}\sum_{n\in V} w(n)}$. The CCR's typical values of 0.1, 1 and 10 represent the low, medium and high communication cases, respectively. The weights of edges are generated randomly from $w_{min} \times CCR$ to $w_{max} \times CCR$.

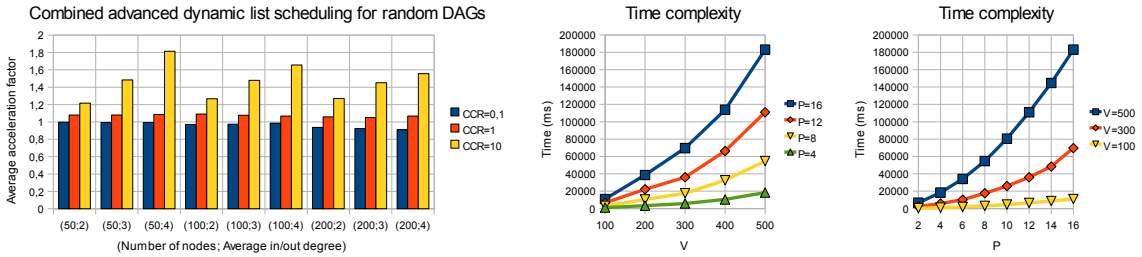
The advanced dynamic list scheduling heuristic can use the five groups of node priorities to get different results. We combine the five groups of node priorities with the advanced dynamic heuristic and choose the best result; the whole process is called a combined advanced dynamic heuristic. To compare the performance difference between the combined advanced dynamic heuristic and the classic list scheduling heuristic with the node priority of bl & tl, we generate random DAGs as follows: The number of nodes is fixed to be 100, weights of nodes vary randomly from $w_{min} = 100$ to $w_{max} = 1000$, and according to the average in/out degree and CCR, 9 groups of random DAGs are generated with 1000 samples in each group. Table 4 compares the combined advanced dynamic heuristic with the classic heuristic. Although the combined advanced dynamic heuristic is worse than classic heuristic for most random DAGs in the case of $CCR = 0.1$, it is better for most random DAGs as the CCR increases.

Table 4: Comparison of the combined advanced dynamic heuristic with the classic list scheduling heuristic

Average in/out degree	2			3			4		
CCR	0.1	1	10	0.1	1	10	0.1	1	10
Better	1.2%	86.4%	94.7%	1.9%	78.2%	95.6%	2.3%	76.6%	95.3%
Equal	24.2%	0.9%	0.0%	13.7%	0.0%	0.0%	8.7%	0.0%	0.0%
Worse	74.6%	12.7%	5.3%	84.4%	21.8%	4.4%	89.0%	23.4%	4.7%

To illustrate more clearly the performance of the combined advanced dynamic heuristic, we define the acceleration factor (Acc) as $Acc = \frac{sl_{classic}}{sl_{advanced}}$ to show the speed-up of the advanced heuristic. We tested 27 groups of random DAGs, and Figure 7(a) shows the average Acc of the combined advanced dynamic list scheduling heuristics. It is noticed that their performances are similar and the schedule results are sped up ($Acc > 1$) by using

the combined advanced heuristic in the cases of $CCR = 1$ and $CCR = 10$. We can see that the average Acc increases when CCR varies from 0.1 to 10. The schedule result can be accelerated up to 80% when $CCR = 10$. If the number of nodes is fixed, the average Acc increases as the average in/out degree increases when $CCR = 10$. The reason for this phenomenon is that the critical child technique helps to select better processors for nodes with multiple predecessors. The greater the in/out degree is, the better the critical child works. Since the communication cost is increasing in modern embedded applications like digital communication and video compression, our method is suitable for scheduling these applications on parallel embedded systems.



(a) Average Acc (b) Time complexity with V (c) Time complexity with P

Figure 7: Average Acc of the advanced dynamic heuristic and its time complexity

5.3 Time Complexity of the Advanced Dynamic Heuristic

The classic list scheduling heuristic has the time complexity of $O\left(PE^2O\left(routing\right)+V^2\right)$, where P, V and E are the number of processors, the number of nodes and the number of edges, respectively. $O\left(routing\right)$ represents the maximum number of links on a route and is usually fixed because of the static routing strategy in parallel embedded systems. The time complexity increases by a factor of P when using the critical child. Hence, the time complexity of our advanced dynamic heuristic is $O\left(P\left(PE^2O\left(routing\right)+V^2\right)\right)$, while combination of the advanced dynamic heuristic with the five node priorities does not increase the degree of the time complexity.

Figure 7(b) and 7(c) shows the time consumed to schedule different sizes of DAGs on architectures with different numbers of processors by our combined advanced dynamic heuristic. All the DAGs have the average in/out degree of 4; all the processors are connected to the same switch by different communication links. It is shown that the time increases with the square of V and also with the square of P. We ran our heuristic on a Pentium Dual-Core PC at 2.4GHz, and it toke about 3 minutes to schedule a DAG with 500 nodes on an architecture of 16 processors. In fact, a complicated embedded application usually has no more than 500 nodes in models of coarse and medium grain, and P is usually much smaller than V and E in a parallel embedded system. Hence, the increase of time complexity is reasonable and acceptable for rapid prototyping methodologies.

6 Conclusions

This paper proposes three new groups of node levels (top level and bottom level) and two advanced techniques (critical child and communication delay) for list scheduling with communication contention. We also give an advanced dynamic list scheduling heuristic using the new node levels and the two advanced techniques. Our method is used for heterogeneous parallel embedded systems. The new node levels take into account the communication contention and are used as node priorities to generate different node lists; the critical child technique helps to select a better processor for a node; and the communication delay technique delays communications when necessary in order to enlarge idle time intervals on communication links.

The advanced dynamic heuristic can use different node lists to get different scheduling results for a given DAG. We combine the five groups of node priorities with the advanced dynamic heuristic and choose the best result; the whole process is a combined advanced dynamic heuristic. To compare with the classic method, we use homogeneous parallel systems and randomly generated DAGs. Experimental results show that the combined advanced dynamic heuristic is efficient to shorten the schedule length for most of the randomly generated DAGs in the cases of medium and high communication. Our method accelerates a scheduling result up to 80% in the

case of high communication and sometimes also reduces the use of hardware resources. Since the communication cost is increasing from low to medium and even to high in modern digital communication and video compression applications, our method will work well for scheduling these applications on parallel embedded systems.

Acknowledgements

This work was supported by the China Scholarship Council. We thank Profs. YIN QinYe of Xi'an Jiaotong University for giving precious propositions during the redaction of this paper.

References

1 Lee E, Parks T. Dataflow process networks, Proceedings of the IEEE, 1995, 83(5): 773–801
2 Sriram S, Bhattacharyya S S. Embedded multiprocessors - scheduling and synchronization. New York, NY, USA: Marcel Dekker, Inc, 2000
3 Sarkar V. Partitioning and scheduling parallel programs for multiprocessors. Cambridge, MA, USA: MIT Press, 1989
4 Garey M R, Johnson D S. Computers and intractability: A guide to the theory of NP-completeness. New York, NY, USA: W H Freeman & Co, 1990
5 Adam T L, Chandy K M, Dickson J R. A comparison of list schedules for parallel processing systems. Commun ACM, 1974, 17(12): 685–690
6 Kasahara H, Narita S. Practical multiprocessor scheduling algorithms for efficient parallel processing. IEEE Trans Comput, 1984, 33(11): 1023–1029
7 Hwang J J, Chow Y C, Anger F D, Lee C Y. Scheduling precedence graphs in systems with interprocessor communication times. SIAM J Comput, 1989, 18(2): 244–257
8 Wu M Y, Gajski D. Hypertool: A programming aid for message-passing systems. IEEE Trans Parallel Distr Syst, 1990, 1(3): 330–343
9 Yang T, Gerasoulis A. DSC: Scheduling parallel tasks on an unbounded number of processors. IEEE Trans Parallel Distr Syst, 1994, 5(9): 951–967
10 Kwok Y K, Ahmad I. Dynamic critical-path scheduling: An effective technique for allocating task graphs onto multiprocessors. IEEE Trans Parallel Distr Syst, 1996, 7(5): 506–521
11 Sih G, Lee E. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. IEEE Trans Parallel Distr Syst, 1993, 4: 175–187
12 Kwok Y K, Ahmad I. Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures. In: Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing, Washington, DC, USA, 1995
13 Grandpierre T, Lavarenne C, Sorel Y. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In: Proceedings of 7th International Workshop on Hardware/Software Co-Design, Rome, Italy, 1999
14 Sinnen O, Sousa L. Communication contention in task scheduling. IEEE Trans Parallel Distr Syst, 2005, 16(6): 503–515
15 Tang X, Li K, Padua D. Communication contention in APN list scheduling algorithm. Sci China Inf Sci, 2009, 52(1): 59–69,
16 Sinnen O. Task scheduling for parallel systems. Hoboken, NJ, USA: John Wiley & Sons, Inc, 2007
17 Kwok Y K, Ahmad I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys, 1999, 31(4): 406–471
18 Sinnen O, Sousa L. List scheduling: Extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. Parallel Computing, 2004, 30(1): 81–101
19 Stuijk S, Geilen M, Basten T. SDF^3: SDF for free. In: Proceedings of 6th International Conference on Application of Concurrency to System Design, Los Alamitos, CA, USA, 2006